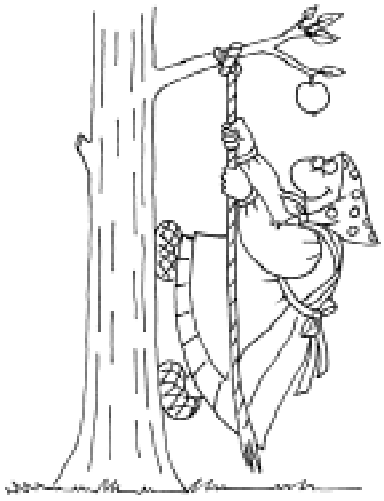
Дмитриева Марина Валерьевна

СПИСКИ И ДРУГИЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

В различных практических приложениях применяются динамические структуры данных, основанные на линейных списках. Рассмотрению некоторых из них (бинарное дерево и стек) посвящена статья. Обсуждается решение задачи обхода для бинарного дерева, приводится рекурсивный вариант решения и обход с помощью стека.

Возможность выделения и освобождения памяти во время выполнения программы позволяет создавать структуры данных с переменным числом элементов, так называемых динамических структур. Среди динамических структур особое место занимают линейные списки, являющиеся основами для других структур данных. Рассмотрим динамическую структуру – дерево, представление дерева с помощью указателей, при решении задачи обхода дерева для обработки вершин используем данные процедурного типа, описанные в предыдущей статье. Другая динамическая структура – стек – может быть представлена с помощью линейного списка. Все структуры данных будут взаимодействовать при решении задачи обхода бинарного дерева с помощью стека.



Решение широкого класса задач можно представить как похождение по дереву.

ИЕРАРХИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Многие реальные объекты имеют иерархическую структуру, например схема предприятия или структура власти в государстве, генеалогическое дерево семьи или родословная некоторого человека.

Для представления таких объектов и обработки связанной с ними информации удобна организация данных, отражающая структуру объектов. Если абстрагироваться от конкретного содержания элементов, то получится математический объект, называемый деревом.

ДЕРЕВЬЯ: ПОСТРОЕНИЕ И ИСПОЛЬЗОВАНИЕ

Многие дискретные процессы принятия решений могут быть представлены в виде дерева. Корень соответствует начальному состоянию, его порожденные вершины соответствуют состояниям, которые могут быть результатом операций из числа имеющихся.

Решение широкого класса задач можно представить как похождение по дереву. Возможность возвращения отображается в дереве как возвращение в предыдущую вер-

шину. Рассмотрим некоторые способы представления и обработки деревьев.

Дерево представляет собой конечное множество элементов, называемых узлами или вершинами. Вершины расположены на разных уровнях иерархии. На самом высоком уровне иерархии (пусть номер этого уровня 1) располагается единственный узел, называемый корнем. Каждый узел, расположенный на **i**-том уровне, связан с единственным узлом на более высоком (**i-1**) уровне. Последний узел является исходным или предком для данного узла.

Каждый узел **i**-того уровня может быть связан с одним или несколькими узлами на более низком (**i+1**) уровне. Такие узлы (**i+1**)-го уровня называются порожденными узлами или потомками. Узлы, не имеющие порожденных, называются листьями. На плоскости узлы удобно изображать точками, узлы **i**-того уровня связывать дугами с порожденными узлами (**i+1**)-го уровня. При работе с деревьями удобно определить пустое дерево как дерево, не содержащее вершин.

БИНАРНЫЕ ДЕРЕВЬЯ И ИХ ПРЕДСТАВЛЕНИЕ

Структуры данных и алгоритмы их обработки будут наиболее простыми в случае так называемых бинарных деревьев, то есть таких деревьев, каждый узел которых имеет не более двух порожденных узлов: левого и правого (и соответственно не более двух поддеревьев: левого и правого). На рис. 1 представлено бинарное дерево.

При таком способе представления достаточно иметь лишь указатель на корень дерева, чтобы получить доступ к любому элементу, спускаясь вниз по указателям. Отсутствующее дерево представляется пустым указателем.

ОПРЕДЕЛЕНИЕ ТИПА БИНАРНОГО ДЕРЕВА

Узел дерева можно представить с помощи записи, состоящей из трех полей, первое поле – информационное, второе и третье – указатели на корни левого и правого поддеревьев. Если элемент имеет тип **node**, то тип значения, называемого указателем на элемент **node**, записывается так: **^node**. В разделе определения типов можно ввести синоним для типа **^node**:

```
tree=^node.
```

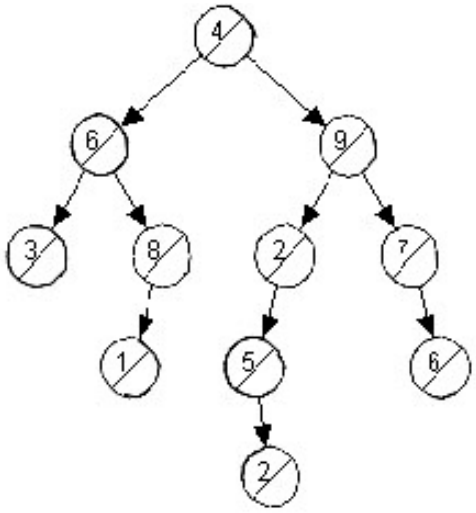


Рис. 1. Бинарное дерево

Описание типов данных для бинарного дерева может выглядеть так (см. листинг 1).

Предположим, что переменная **t** описана так: **var t: tree**. Напомним, что переменная, на которую установлен указатель **t**, обозначается **t^**, в нашем случае **t^** имеет тип node. Получить доступ к полям переменной **t^** можно следующим образом:

```
t^.info, t^.left, t^.right.
```

ЗАДАЧА ОБХОДА ДЕРЕВА. РЕКУРСИВНЫЙ ВАРИАНТ

Одна из распространенных задач при работе с деревьями: выполнение некоторо-

Листинг 1. Описание данных для бинарного дерева

```
type elem_tree = integer;
     tree = ^node;
     node = record info: elem_tree;
            left, right: tree
  end
```

Листинг 2. Обход дерева с заданной процедурой обработки вершины

```
procedure trav_l (q: tree; p: action);
  begin
    if q<>nil
    then
      begin
          p(q^.info);{обработка корня  с помощью процедуры P}
          trav_l(q^.left,p); {обход левого поддерева}
          trav_l(q^.right,p); {обход правого поддерева}
      end
    end
```

го действия **P** с каждым элементом дерева. Часто такую задачу называют задачей обхода дерева.

В качестве примера приведем обход дерева:

– действие **P** применяется к корню дерева,

– осуществляется обход левого поддерева корня (если поддерево существует),

– осуществляется обход правого поддерево (если поддерево существует).

В листинге 1 содержится описание процедуры обхода дерева, каждый элемент дерева просматривается один раз. Второй параметр задает действие, выполняемое над каждым элементом дерева. Для этого необходимо определить процедурный тип данных, например, таким образом

```
type action= procedure (var z: elem_tree)
```

Описание процедуры обхода дерева приведено в листинге 2.

Пусть процедура **out_info** помещает значение параметра в стандартный файл вывода, а процедура **mul10** увеличивает значение параметра в десять раз. Описание процедур в листинге 3.

При выполнении вызова **trav_l (q, out_info)** при обходе дерева значения информационных вершин дерева поступят в стандартный файл вывода, при выполнении вызова **trav_l (q, mul10)** значение каждой информационной вершины будет увеличено в десять раз. На рис. 2 представлен результат обхода бинарного дерева с помощью описанной процедуры. В каждой вершине число над чертой является информационным полем дерева, а число под чертой определяет порядок обработки вершин.

Одной из основных операций при работе со сложными структурами данных является операция построения этой структуры. Рассмотрим алгоритм построения бинарного дерева по последовательности символов.

... такую задачу называют задачей обхода дерева.

Листинг 3. Описание процедур обработки элементов списка

```
{$F+}
procedure out_info (var z: elem_tree);
  begin writeln (z,'')  end;
procedure mul10 (var z: elem_tree);
  begin z := 10*z end;
{$F-}
```

Листинг 4. Построение бинарного дерева по линейной скобочной записи

```
Procedure  cr_tree ( var q: tree);
  Var c: char;
   Begin
      c := cursym;
      c := cursym;
      if c  = ')' then  q := nil
      else
         begin
           new (c); {создание корня дерева}
           q^.info := c;
           cr_tree (q^.left); {построение левого поддерева}
           cr_tree (q^.right); {построение правого поддерева}
         end
   end
```

ЛИНЕЙНАЯ СКОБОЧНАЯ ЗАПИСЬ

Для задания дерева в виде строки используется так называемая линейно-скобочная запись. Рассмотрим следующую линейно-скобочную запись: двумя скобками `()` задается пустое дерево, дерево, содержащее лишь корень `A`, задается строкой `(A()())`, дерево с корнем `A` и двумя поддеревьями с корнями `B` и `C` задается записью `(A(B()())(C()()))` и т.д. В листинге 4 приведено описание процедуры, которая по линейной скобочной записи строит бинарное дерево. Предполагается в данном случае, что тип элементов информационного поля `char`.

Функция без параметров cursym выбирает для анализа очередной символ строки, представляющей линейную скобочную запись дерева. Она может быть описана, например, так, как в листинге 5.

Перейдем к рассмотрению еще одной структуры данных, реализация которой может быть тесно связана с линейными списками.

СТЕК И ЕГО РЕАЛИЗАЦИЯ С ПОМОЩЬЮ СПИСКА

Будем называть стеком последовательность элементов одного и того же произвольного типа, к которой можно добавлять новые элементы и убирать элементы из этой последовательности, причем как добавление новых элементов, так и удаление старых производится из одного и того же конца этой последовательности, называемой вершиной стека. Иначе говоря, при удалении элементов всегда удаляется тот элемент, который был последним добавлен в стек, таким образом, элементы удаляются в порядке, обратном порядку добавления эле-

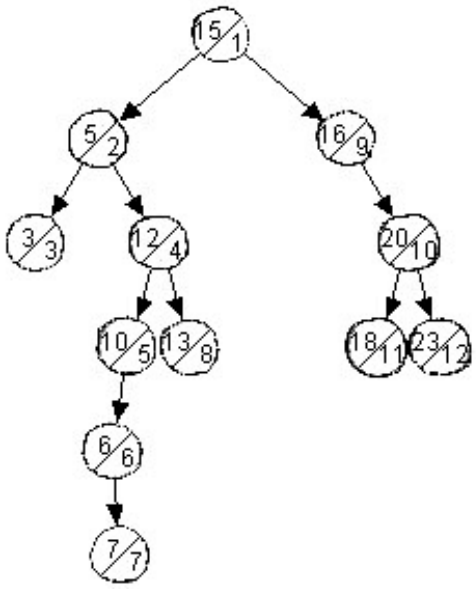


Рис. 2. Рекурсивный обход бинарного дерева

Листинг 5. Функция выбора для анализа очередного символа

```
function cursym: char;
  var  l : char;
begin read (l); cursym := l end
```

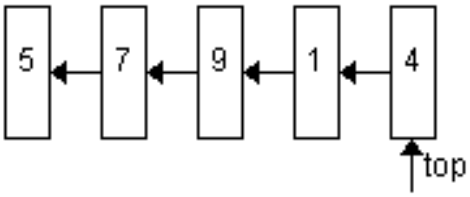


Рис. 3. Представление стека с помощью списка

ментов. Зададим набор операций, характеризующий стек как тип данных:

– **initstack(s)** – операция создания нового стека **s**;

– **push (s,e)** – операция добавления (заталкивания) элемента e в стек **s**;

– **pop (s)** – операция удаления (выталкивания) элемента из стека **s**. Часто удобно, чтобы эта операция возвращала выталкиваемый элемент в качестве результата;

– **top (s)** – операция, выдающая верхний элемент стека **s**. Остальные элементы стека, как правило, непосредственно не доступны;

– **empty (s)** – операция проверки пустоты стека.

Приведем реализацию стека таким образом, что все его элементы связываются в список, первым элементом которого служит вершина стека. Весь список, а значит и сам стек, представлен указателем начала списка. На рис. 3 представлен стек, реализованный в виде линейного списка.

Описание данных для стека приведено в листинге 6.

Приведем реализацию операций над стеком в случае, когда стек представлен линейным списком (листинг 7).

Операция добавления в стек реализуется вставкой элемента списка в начало (листинг 8).

Удаление элемента из стека соответствует удалению первого элемента списка. Информационное поле удаляемого элемента является результатом работы функции. При выталкивании элемента из стека память, занимаемая этим элементом, освобождается (листинг 9).

Элементом на вершине стека является информационное поле первого элемента списка (листинг 10, 11).

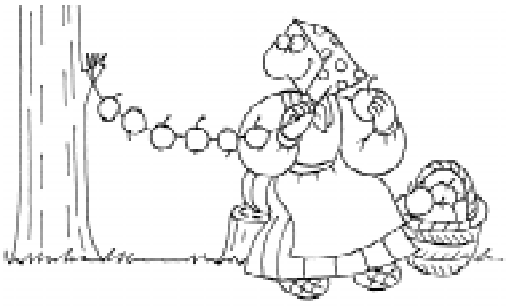
При решении задач, использующих стек, целесообразно описать модуль для работы со стеком и затем подключить его к программам.

Листинг 6. Реализация описания данных для стека

```
type elem_stc k= integer;
     stack=^elem;
     elem= record info: elem_stck;
                  next: stack
           end;
```

Листинг 7. Реализация операции инициализации

```
procedure initstack (var s: stack);
      begin s:= nil end;
```



Будем называть стеком последовательность элементов одного и того же произвольного типа...

Листинг 8. Реализация операции добавления элемента в стек

```
procedure push (var s: stack; r: elem_stck);
        var p : stack;
      begin new (p);
            if p=nil then error ('Переполнение стека');
            p^.info := r;
            p^.next := s;
            s:= p
      end;
```

Листинг 9. Реализация операции удаления элемента из стека

```
function pop (var s: stack): elem_stck;
         var p : stack;
      begin if s= nil then error ('Исчерпание стека');
            pop := s^.info;
            p := s;
            s := s^.next;
            dispose (p)
      end;

      end;
```

Листинг 10. Реализация операции «элемент на вершине стека»

```
function top (var s: stack): elem_stck;
       begin if s= nil then error ('Исчерпание стека');
             top := s^.info
       end;
```

Листинг 11. Реализация операции «проверка, не является ли стек пустым»

```
     function empty (var s: stack): boolean;
       begin empty := s=nil end;
```

ИСПОЛЬЗОВАНИЕ СТЕКА ПРИ АНАЛИЗЕ ЛИНЕЙНОЙ СКОБОЧНОЙ ЗАПИСИ

Обсудим решение более общей задачи анализа скобочной структуры текста. Пусть задан некоторый текст, содержащий три типа скобок: ( ), [ ] и { }. Необходимо проверить правильность расстановки скобок в тексте, то есть выяснить, имеется ли для каждой открывающей скобки соответствующая закрывающая скобка и наоборот, а также не нарушены ли правила вложенности скобок. Символы, не являющиеся скобками, игнорируются. Алгоритм анализа скобочной структуры текста может быть следующим:

- Встретив открывающую скобку, помещаем ее в стек.
- Обнаружив закрывающую скобку, проверяем, находится ли на вершине стека соответствующая открывающая скобка. Если проверка была успешной, то после проверки скобка удаляется из стека, иначе программа заканчивает работу, выдав отрицательный результат.
- Если в исходном тексте закрывающих скобок оказывается больше, чем открывающих, то программа попытается извлечь символ из пустого стека и закончит работу с отрицательным результатом.
- Наоборот, если в тексте открывающих скобок окажется больше, чем закрывающих, то, после того как весь текст будет прочитан и обработан, в стеке еще останутся «незакрытые» скобки.
- Таким образом, результат проверки текста окажется успешным, если во время работы ни разу не обнаружится несоответствия скобок, а также не возникнет ситуации исчерпания стека, причем в конце обработки текста стек окажется пустым.

При построении линейной скобочной записи в предложенном варианте используется один тип скобок, поэтому и алгоритм анализа будет проще описанного.

ИСПОЛЬЗОВАНИЕ СТЕКА ПРИ ОБХОДЕ ДЕРЕВА

Если рассматривать дерево как абстрактный тип данных, предназначенный для хранения и обработки однородной информации, то следует определить набор основных операций над деревом. В число таких операций может входить включение в дере-

...поиск элементов в дереве удается организовать быстрее, если элементы в нем хранятся упорядоченно.

во нового элемента или исключение одного из элементов дерева, определение принадлежности элемента дереву, то есть поиск заданного элемента и т. п. Реализация всех этих операций зависит не только от способа представления дерева, но и от порядка расположения элементов в нем, например поиск элементов в дереве удается организовать быстрее, если элементы в нем хранятся упорядоченно.

Рассмотрим операцию, не зависящую от содержания дерева и порядка расположения элементов в нем. Такой операцией является обход дерева, то есть выполнение некоторого действия для каждой вершины этого дерева. Рекурсивный вариант реализации обхода дерева был рассмотрен ранее.

В некоторых случаях рекурсивный вариант итерации оказывается неприемлемым из-за чрезмерных накладных расходов. Напишем не рекурсивную процедуру, осуществляющую обход дерева сверху вниз, при котором сначала необходимое действие применяется к корню дерева, а затем по очереди обходятся сверху вниз левое и правое поддеревья корня.

В нерекурсивной процедуре локальная переменная cur будет содержать указатель, последовательно указывающий на все вершины дерева. При этом необходимо запоминать, какие части дерева остались еще не пройденными. Удобно использовать для этого стек, хранящий указатели на вершины дерева.

Алгоритм будет действовать следующим образом. Вначале в стек помещается указатель на корень дерева. Это означает, что пока еще все дерево необходимо просматривать. Далее в цикле извлекается из стека указатель на вершину, применяется к этой вершине нужное действие, а затем помещается в стек для последующего обхода правое и левое поддеревья этой вершины (если они есть). В этом случае будет достигнут необходимый порядок обхода вершин дерева.

Элементами стека при решении этой задачи должны быть указатели на вершины дерева:

```
type elem_stc k= tree.
```

На рис. 4 изображено дерево и состояние стека при его обходе.

Тип **action**, как и раньше, определяет процедурный тип данных, процедуры такого типа будут использоваться для обработки вершин дерева при обходе дерева. Описание процедуры обхода дерева с помощью стека представлено в листинге 12.

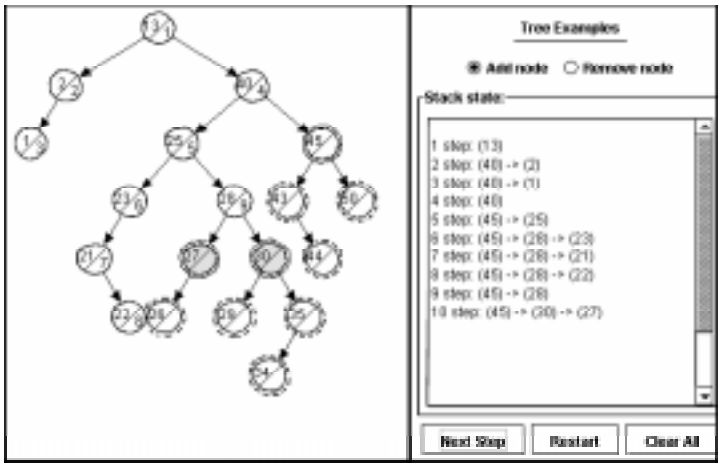


Рис. 4. Обход дерева с помощью стека

При обходе дерева можно, например, формировать список. Обработка вершины дерева состоит в добавлении в линейный список элемента с информационным полем обрабатываемой вершины. При обходе дерева, изображенного на рис. 5, будет построен список, изображенный на рис. 6.

Элементы добавляются в конец списка.

Листинг 12. Процедура обхода дерева с помощью стека

```
procedure trav_tree (t: tree; p: action);
   var cur: tree; {обрабатываемая вершина дерева}
   s: stack; {стек, хранящий указатели на необработанные вершины
дерева}
begin
   if t = nil then exit;
   initstack (s); {инициализация стека}
   push (s,t); {помещение в стек указателя на корень дерева}
   repeat cur := pop (s);{извлечение из стека очередного указателя}
          p (cur^.info);{обработка вершины дерева}
          if cur^.right <> nil then push (s, cur^.right);
            {помещение в стек указателя на корень правого дерева}
          if cur^.left <> nil then push (s, cur^.left)
            {помещение в стек указателя на корень левого дерева}
    until empty (s)
end
```

Мы рассмотрели механизм итерации на примере бинарных деревьев. Напоминаем, что основная идея механизма итерации состоит в отделении информации о внутреннем устройстве сложной структуры данных (списка, дерева) от процедур обработки его элементов.

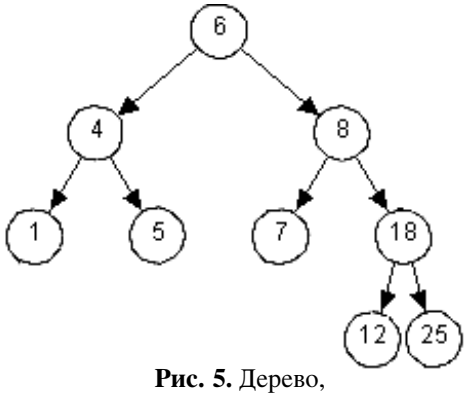


Рис. 5. Дерево, при обходе которого строится список

ЗАДАЧИ

1. Напишите программу, которая при обходе бинарного дерева строит линейный список.

2. Напишите программу, которая при обходе дерева определяет элемент с максимальным значением информационного поля.

3. Напишите программу, которая при обходе бинарного дерева строит упорядоченный линейный список.

4. Напишите программу, которая проверяет правильность записи линейно-скобочной структуры дерева с использованием стека.

5. Опишите процедуры, которые осуществляют обход бинарного дерева снизу вверх, представьте рекурсивный вариант и с использованием стека. Обход снизу вверх предполагает обход левого поддерева, обход правого поддерева, обработку корня.

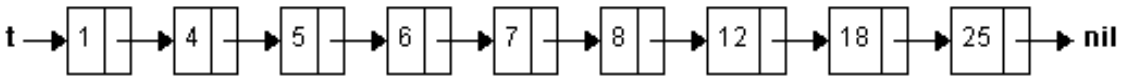


Рис. 6. Список, построенный при обходе дерева

Дмитриева Марина Валерьевна, доцент кафедры информатики математико-механического факультета Санкт-Петербургского государственного университета.